Journal of Universal Computer Science, vol. 15, no. 4 (2009), 898-925
submitted: 30/6/08, accepted: 27/2/09, appeared: 28/2/09 © J.UCS

SDLMAS: A Scenario Modeling Framework for Multi-Agent Systems

Igor Čavrak
(University of Zagreb, Croatia
igor.cavrak@fer.hr)

Armin Stranjak
(Strategic Research Centre, Rolls-Royce plc, Derby, UK
armin.stranjak@rolls-royce.com)

Mario Žagar
(University of Zagreb, Croatia
mario.zagar@fer.hr)

Abstract: In this paper we analyze existing methods and languages for modeling agent interactions and propose a SDLMAS Framework for rapid design, development and runtime support of multi-agent systems. The framework provides a simple but expressive declarative language for modeling complex interactions among agents. Proposed language is based on scenarios, sequences of conversation actions directed towards achieving a goal. Scenario descriptions are converted into program code for a chosen target agent platform and system execution is supported by a runtime framework.
Key Words: Multi-Agent System, Interaction, Scenario, Language
Category: D.2.0, I.2.5, I.2.11, I.6.0

1 Introduction

The development of computer networks in recent years has enabled exponential expansion of distributed information and data accessible through increasing number of computer systems. As a consequence, inevitable requirements have emerged for new software methodologies which allow seamless and transparent access to the expanding amount of information. The principles of agent-centric distributed computing show significant potential to deal with such necessities. This new paradigm advances the typically modular, client-server approach by introduction of concepts like autonomy, interaction and social behavior, to name a few [Jennings et al. 99]. In such systems, agents fulfill own goals autonomously by sensing their environment and by interacting with other agents exchanging data if and when it is in their interest. This context-dependent behavior relies heavily on the characteristics of the environment.

Competitive environment encourages agents to act within market-based framework where they compete with each other for better price, more computing power, shorter service response, etc. An agent's intentions are predominantly influenced by its own goals and plans to accomplish them. Alternatively, cooperative environment promotes such agents that will perform their goals in the interest of the wider community or the authoritative entity that secures the fulfillment of the global goal. Typical examples would be applications for task planning and resource scheduling, search engines, or any other where the emergent behavior is influenced by collaborative and mutually non-exclusive individual goals.

Irrespective of the types of agent environment, communication between them is achieved through asynchronous and message-oriented interactions. In addition to physical connection, it requires semantics in order to enable agents to reach a common understanding. Therefore, communication protocols and corresponding dialogue ontology need to be defined too within the framework of a conversation space. This space is defined as a sequence of messages exchanged between agents following a (set of) defined dialogue protocol(s). Protocols allow agents to participate in conversations by prescribing shared protocol semantics and defining conversation space within which agents are enabled to act, still preserving their decision-making autonomy [Greaves et al. 00]. This space is defined by the following: (1) an ontology of common terms that agents need to agree upon, (2) a language for ontology description, and (3) a language for a conversation description.

In this context, we can identify several conversation models in existing literature. The model described in [Bratman et al. 88] is based on a belief-desire-intention (BDI) scheme, but it suffers from the "semantic verification" problem [Wooldridge 00]. The MAP [Walton 03] language offers concepts of scenes and roles in multi-agent systems. Although it introduces an interesting concept of scenes and roles, its agent-centric nature increases complexity of conversation verification where inconsistent and unstructured protocols are not easy to detect. The idea of protocol being defined within an agent's own business logic is described in [Endriss et al. 03]. The same idea was adopted by the IOM/T [Doi et al. 05] language where main focus is on message flow between agents, rather than agent's activities like MAP. Unfortunately, the language relies on a particular agent platform which limits its use on other agent platforms and hence large-scale adoption.

This paper introduces a new language, SDLMAS, for scenario description in multi-agent systems, independent of the target platform and implementation language. It adopts an interaction-centric approach with focus on message flow between agents providing intuitive scenario description. The paper also focuses on an SDLMAS platform for rapid development of agent-based systems by enabling automatic code generation from the SDLMAS language into a target implementation language and an agent platform. The platform allows scenario development and execution by providing a runtime framework and placeholders for agent's internal business logic, leaving complexity of interactions and message propagation hidden from the developer, thereby letting him/her concentrate on agent functionalities.

The initial motivation for the development of the SDLMAS language was support of multi-agent simulation system for prediction and scheduling of engine overhaul in aerospace industry [Stranjak et al. 08]. Complex scenario descriptions of interactions between engine fleet managers and overhaul bases required significant design and development efforts. Such difficult problem would require a declarative language for conversation framework definition within which simultaneous interactions would occur without explicit specification of their ordering or timing. In addition to this, a requirement to define intertwining protocols was necessary to enable cross-scenario communication. In other words, this would allow agents to achieve given tasks or gather required information within one scenario and to communicate them to the other. These agents would need to maintain several conversations simultaneously during negotiations for the best shop visit time and repair slot while utilizing balance between revenue earned from the engines in service and an acceptable risk of disruption. Current scenario description languages cannot satisfy given requirements and therefore it was necessary to define a new interaction description language and build a corresponding platform.

The SDLMAS platform was created in order to equip a designer with the facility to describe negotiation scenarios and a developer to concentrate only on agent's business logic by generating necessary helper classes to support message passing, execution of scenario actions and invocation of given business procedures. This way, the development cycle was shorten several times with significantly increased reliability of the system stability after deployment.

The SDLMAS language and platform are applicable not only in closed multi-agent systems such as a specialized simulation system described above, but are suitable for a wide variety of MAS applications involving moderate-to-complex interactions, such as e-commerce [Papazoglou 01] and supply-chain management systems and simulations [Podobnik et al. 08], information retrieval systems [Sliwko et al. 07], agent based recommender systems [Zhang et al. 08] etc. With the future incorporation of fine-grained security mechanisms, the SDLMAS platform will gain even larger potential for its application in open multi-user systems.

2 Scenario Description Language

2.1 Language properties

The SDLMAS scenario description language, together with the SDLMAS runtime framework, represents the core component of the SDLMAS platform. The language has been designed with the main purposes of facilitating and accelerating design and development phases of multi-agent systems. In order to fulfill those goals three main language properties had to be achieved:

– descriptions of interactions among agents in the multi-agent system must be simple and intuitive, yet scalable enough to allow description of complex agent interactions,

– the language must be expressive enough to allow definition of a strict but flexible interaction space within the described multi-agent system,

– the language must, as much as possible, protect its users from run-time complexities related to interaction management within a large multi-agent system.

The SDLMAS language is a declarative, interaction-centric description language, focused on defining allowed sequences of messages (communicative acts) exchanged among interaction participants (agents in a multi-agent system). In SDLMAS an interaction protocol is defined implicitly, as a sequence of agent actions where order of those actions is important, thus achieving a sequential approach in protocol definition. The language describes conversations among agents as a sequence of conversation actions, where actions define a conditional mapping between incoming and outgoing messages, and an agent's internal logic. The language explicitly defines conditions for reception and transmission of messages as part of a conversation protocol definition. An elementary action of the language is defined as a procedure that will be executed as a consequence of a condition being satisfied following a message reception. A scenario is formed of a logically complete sequence of conversation actions aimed at achieving some rational effect in the multi-agent system.

The language is restrained to communication aspect of multi-agent systems, thus enforcing strict separation between conversation actions and internal agent logic implementing agent reasoning. Although direct influence on agent decision process is not possible, inadequate expressiveness could indirectly restrict or bias the way agent reasons or acts, or simply fail the attempt to model interactions of required complexity. The SDLMAS language provides adequate expressiveness by:

– allowing parallelism in agent interactions, defining synchronization points,

– defining conditions on incoming and outgoing messages in conversation actions, including complex logical expressions containing multiple message types and agents as sources or targets of incoming and outgoing messages,

– allowing controlled variations in message exchanges during scenario execution, effectively defining a (possibly infinite) set of allowed interaction scenarios from one scenario definition,

– generalizing interaction patterns (scenarios) to roles, not restricting them to particular agents.

Runtime interaction complexities within non-trivial multi-agent systems are predominantly caused by:

– a need to support possibly many scenario instances the agent can concurrently participate in,

– a need to handle parallel conversations with possibly many agents within the same scenario instance, and to ensure conversation adherence to scenario defined rules,

– a need to maintain conversation state with a particular agent within a scenario instance and correctly terminate conversations

– a need to correctly handle exceptions during conversation, both at the execution and at the protocol level.

Most of the mentioned complexities are hidden from the system designer and developer by the platform run-time mechanisms, although the language itself provides several features, including language elements for representation and simultaneous communication with a variable group of agents sharing the same role.

Due to its simplicity, declarative nature of scenario definitions and strict separation of communication aspects of multi-agent system from agents' internal implementations, developed scenarios are completely independent of the agent platform and implementation language. As such, they allow relatively simple analysis and consistency validation, as well as transformation into alternative models. As the final step in the process of modeling and designing interactions within a multi-agent system, defined interaction scenarios are transformed into a program code for target agent platform and implementation language.

2.2 Example scenario

The example system depicted on [Fig. 1] represents a set of electrical power producers and consumers within an autonomous vehicle. All of the producers and consumers are abstracted with corresponding agents and form a virtual energy market. The aim of the system is to optimize energy usage patterns within a vehicle by modeling power source availability and characteristics, and to adjust power usage policies of power consumers to such characteristics. The intent of this description is not to focus on the described system's architecture or functionality, but to provide a context for presenting all of the key elements of the SDLMAS language and platform.

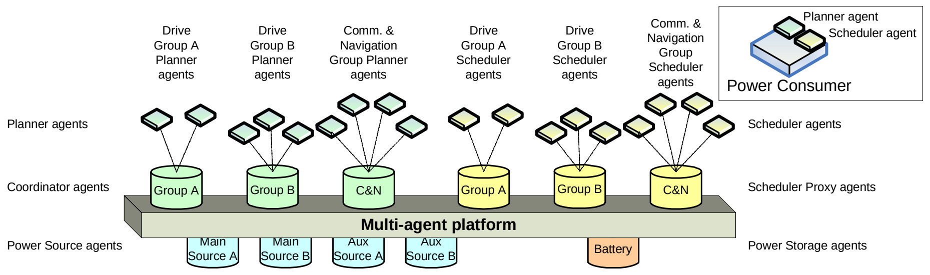


Figure 1: Example multi-agent system of an autonomous vehicle

Each power consumer within the described system is represented by two agents: a planner agent, in charge of future power consumption planning and reservation, and a scheduler agent, scheduling consumer actions on the basis of acquired power reservations. Consumers are grouped and managed by group coordinators represented by two agent types: a `Coordinator` agent type and a `SchedulerProxy` agent type. `Coordinator` agents, among other duties, hold pre-allocated power reservations from active power sources and distribute it among coordinated consumers, and serve consumer requests for additional power reservations. `SchedulerProxy` agents negotiate power requests originated from `Coordinator` agents, mediate power reservation changes with `Scheduler` agents within their coordination group, and sell power surpluses. Agents of type `PowerSource` and `PowerStorage` abstract various power sources and their characteristics. The `ConsumerPowerRequest` scenario depicts an interaction where an agent of type `Planner`, representing an electric drive 3, requests additional amount of power for a specified time period in order to perform its task. This power is requested from the agent's power consumer group coordinator, who can employ two strategies. A requested amount of power can immediately be allocated to `drive3` if the sufficient amount of pre-allocated power has been available to the coordinator, or the power can be requested from other system components. As the first step towards obtaining additional power the coordinator issues a call for proposals to all of the other coordinators in the system (represented with `SchedulerProxy` agents). In turn, scheduler proxies also have two options at their disposal: offer their pre-allocated power for a suitable compensation or issue a call for proposals to all of the consumers (represented with `Scheduler` agents) in the group they coordinate. In both cases, the offer is returned or a refuse message sent on the basis of availability of pre-allocated coordinator power or willingness of consumers within a coordinated group to release (sell) their power reservations. After all the offers and refusals are collected from peer coordinators, a call for proposal is issued by the coordinator

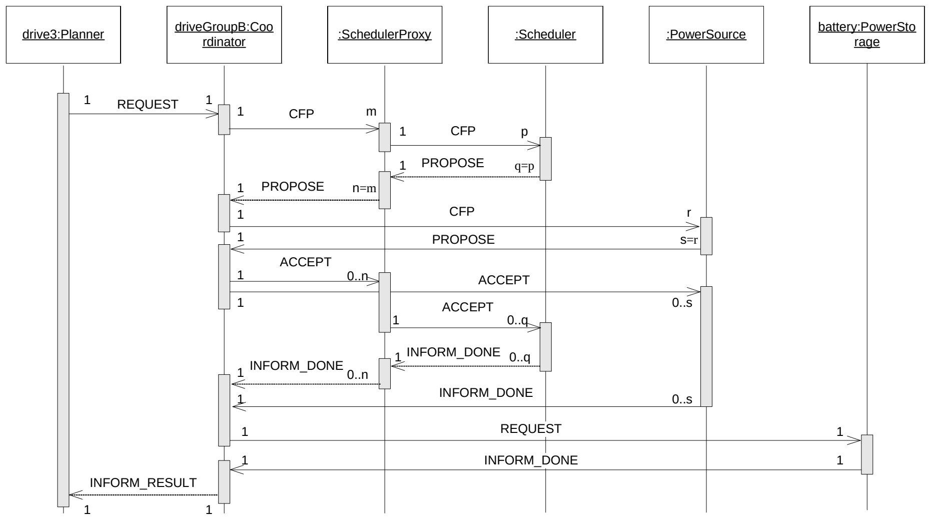


Figure 2: Sequence diagram of the ConsumerPowerRequest interaction scenario

to all of the system's active power sources. Collected offers are evaluated on the basis of their temporal characteristics and compensation requested from the originating source, and the most suitable one is selected. If no satisfactory offer has been received, or no offers have been received at all (in case of high system load), the coordinator can, as the last resort, request additional power from the `battery` (single power source of such a type in the system). The result of this complex interaction among the system's actors is finally relayed to the requesting drive3 `Planner` agent. [Fig. 2] contains a very simplified sequence diagram of the described interactions.

The key characteristic of the presented interaction that makes sequence diagrams less suitable for describing it is the intertwining of three contract net protocols - results from one protocol have influence on execution of other protocols, yielding a large amount of possible variations within the execution of the scenario. Also, there are large sections of parallel conversations among agents and optional executions of certain scenario parts (most of the scenario can be omitted as the coordinator is able to allocate power from its internal reserves).

2.2.1 Example Scenario Definition

This section contains the complete source of the `ConsumerPowerRequest` scenario defined in the SDLMAS language.

```
agent @planner                   : Planner;
agent $coordinator, @coordinator : Coordinator;
agent @schedulerProxy            : SchedulerProxy;
agent $localSchedulers           : Scheduler;
agent source                     : PowerSource;
agent battery                    : PowerStorage;

scenario ConsumerPowerRequest {

 action Planner in issuePwrAllocationRequest() {
   msgSnd : (REQUEST -> $coordinator);
 }

 action Coordinator in receivePwrAllocationRequest() {
   msgRcv : (REQUEST <- @planner);
   msgSnd : (CFP -> <SchedulerProxy>) |
            (#INFORM_RESULT -> @planner);
 }

 action SchedulerProxy in receivePwrReleaseProposal() {
   msgRcv : (CFP <- @coordinator);
   msgSnd : (CFP -> <$localSchedulers>) |
            (PROPOSE | #REFUSE -> @coordinator);
 }

 action Scheduler in receivePwrReleaseProposal() {
   msgRcv : (CFP <- @schedulerProxy);
   msgSnd : (PROPOSE | #REFUSE -> @schedulerProxy);
 }

 action SchedulerProxy in collectPwrReleaseProposals() {
   msgRcv : (PROPOSE | #REFUSE <- <$localSchedulers>);
   msgSnd : (PROPOSE | #REFUSE -> @coordinator);
 }

 action Coordinator in collectPwrReleaseProposals() {
   msgRcv : (PROPOSE | #REFUSE <- <SchedulerProxy>);
   msgSnd : (CFP -> <PowerSource>);
 }

 action PowerSource in processPwrRequestProposal() {
   msgRcv : (CFP <- @coordinator);
   msgSnd : (PROPOSE | #REFUSE -> @coordinator);
 }

 action Coordinator in decideOnPwrAllocationSource() {
   msgRcv : (PROPOSE | #REFUSE <- <PowerSource>);
   msgSnd : (ACCEPT | #REJECT -> <SchedulerProxy>) &
            (ACCEPT | #REJECT -> <PowerSource>);
 }

 action PowerSource in allocatePower() {
   msgRcv : (ACCEPT | #REJECT <- @coordinator);
   msgSnd : (#FAILURE | #INFORM_DONE -> @coordinator);
 }

 action SchedulerProxy in releasePower() {
   msgRcv : (ACCEPT | #REJECT <- @coordinator);
   msgSnd : (ACCEPT | #REJECT -> <$localSchedulers>);
 }

 action Scheduler in confirmPwrRelease() {
   msgRcv : (ACCEPT | #REJECT <- @schedulerProxy);
   msgSnd : (#FAILURE | #INFORM_DONE -> @schedulerProxy);
 }

 action SchedulerProxy in collectConsumerCommits() {
   msgRcv : (#FAILURE | #INFORM_DONE <- <$localSchedulers>);
   msgSnd : (#FAILURE | #INFORM_DONE -> @coordinator);
 }

 action Coordinator in collectPwrAllocationCommits() {
   msgRcv : (#FAILURE | #INFORM_DONE <- <SchedulerProxy>) &
            (#FAILURE | #INFORM_DONE <- <PowerSource>);
   msgSnd : (#REFUSE | #INFORM_RESULT -> @planner) |
            (REQUEST -> battery);
 }

 action PowerStorage in receivePwrRequest() {
   msgRcv : (REQUEST <- @coordinator);
   msgSnd : (#REFUSE | #INFORM_DONE -> @coordinator);
 }

 action Coordinator in receiveBatteryResponse() {
   msgRcv : (#REFUSE | #INFORM_DONE <- battery);
   msgSnd : (#REFUSE | #INFORM_RESULT -> @planner);
 }

 action Planner in receivePwrAllocationResponse() {
   msgRcv : (#REFUSE | #INFORM_RESULT <- $coordinator);
 }
}
```

2.3 Language description

In SDLMAS language, interaction among agents in a multi-agent system is defined in the form of sequential descriptions of conversation actions - scenarios. A dialog is defined as a set of agent reference type declarations and a set of scenarios. A typical SDLMAS dialog definition is a text file consisting of a header part with agent declarations and a body part containing one or more scenario definitions.

2.3.1 Roles and Scenarios

SDLMAS considers roles as standardized patterns of behavior required of all agents participating in conversations conforming to a set of scenarios the system behavior is defined with. Roles are not defined explicitly, but implicit definition is present in the form of a role's conversation actions of all scenarios a particular role participates in. A concrete, explicit role definition is embodied in a generated program code for each of the roles defined in the system. Line 1 of the example scenario contains a declaration of an agent reference, used later in the scenario definition, and its `Planner` type (role). `Planner` role behavior is defined by `issuePwrAllocationRequest` (line 10) and `receivePwrAllocationResponse` (line 89) conversation actions.

Following declarations of agent references and their types (roles), a number of interaction scenarios are defined. A complete set of scenarios for a multi-agent system describes all valid interactions among its constituent parts and defines an external behavior of that system. Each scenario is identified by a unique scenario name and built of a sequence of conversation actions, implicitly defining an interaction protocol. An individual scenario should focus on addressing only a part of system's interactions, aimed at some particular goal the system is striving to accomplish using that scenario. In the example presented in this chapter, only one scenario is defined, although it is clear that many more scenarios are necessary in order to completely describe its functionality and external system behavior.

Time ordering of conversation actions is only partially defined by their relative position in scenario definition. For example, actions `issuePwrAllocation-Request` (line 10) and `receivePwrAllocationRequest` (line 14) are guaranteed to be executed by agents playing respective roles in order in which they are defined in the scenario. The execution order of actions `allocatePower` (line 52) and `releasePower` (line 57) cannot be guaranteed as they belong to separate (parallel) conversations, all that can be guaranteed is that those actions will be executed after `decideOnPwrAllocationSource` (line 46).

2.3.2 Conversation actions

A conversation action is defined within the scope of a scenario, belongs to a role and consists of a procedure π (connector to internal agent logic) and two elementary communication operations: message reception and transmission. Action execution implies that:

– received messages satisfy defined message reception conditions (communicative preconditions) ρ,

– an internal agent's procedure π is executed,

– all messages generated by internal agent's procedure conform to message transmission conditions (communicative postconditions) σ.

Agent's internal procedure π will not be invoked until all of the expected messages arrive and agent is in the adequate conversation state. In case of communication timeout, a special message (containing terminating performative) is inserted in place of an expected one and the procedure is invoked. Three types of conversation actions exist in scenarios:

– Scenario triggering action definition does not define message reception operation. As the first action in a scenario, there are no messages to be received (line 10).

– Regular action is defined with all three action elements, and is characterized by a typical reactive behavior (line 26).

– Conversation terminating action characterized by absence of a message transmission operation, and is usually the last action in scenario (line 89).

2.3.3 Action conditions

Message reception and transmission conditions are defined with respect to message performative(s) and message originating agent(s). A message reception condition ρ of an action a defines circumstances under which a received message or a set of received messages is to be passed to a procedure implementing internal agent logic. A condition consists of a list of expected message performatives and their originating agents, and can form complex expressions using logical operators. Definition of message transmission condition σ is similar to the previously described reception condition ρ. This condition describes circumstances (i.e. message performatives) under which a message or a set of messages, resulting from an execution of an internal agent logic procedure, will be sent to corresponding agents.

A simple communicative action condition is formed of an atomic communicative condition consisting of a required message performative and a target or source agent reference (examples on lines 11 and 15). A more flexible atomic condition definition is allowed by stating a list of required performatives separated by the | operator (operator *or*), effectively achieving 'one of performatives' semantics (lines 23 and 32).

Complex conditions are formed of a number of atomic conditions combined using logical operators *or* and *and* (| and &) in conjunction with parenthesis as grouping operator (lines 22 and 23). The complexity of such conditions is not limited by the language constructs, but only by their logical consistency. The ability to define compound conditions for reception of messages enables system designers to describe complex and concurrent conversations among many agents, while retaining a strict control over passing received messages to internal agent logic.

Sequential nature of conversations among agents can be described using only simple conditions, and stems from the very nature of sequential conversation description in SDLMAS. In the example scenario, `driveGroupB` coordinator first contacts `SchedulerProxy` agents (line 16), and only after all of the responses are collected (line 37) proposals from `PowerSource` agents are requested (line 38).

Expressing parallelism in conversations among agents of different types requires usage of complex conversation conditions. For example, `driveGroupB Coordinator` accepts or rejects `SchedulerProxy` and `PowerSource` proposals in parallel (lines 48 and 49), effectively defining a 'conversation span' point (`decideOnPwrAllocationSource`) and a corresponding 'conversation join' point in action `collectPwrAllocationCommits` (lines 73 and 74). Further, conversations can gain parallelism even more, as new conversation threads span following the scenario definition (lines 22 and 32 as implicit conversation span and join points, where `SchedulerProxy` agent contacts all `Scheduler` agents in its group).

Portions of scenario can be made optional by using *or* operators in defining conversation conditions. For example, the complex outbound condition on lines 16 and 17 permits scenario to end without involving agents other than `drive3` and `driveGroupB` in conversation at all. If message `#INFORM_RESULT` is sent as a result of execution of procedure `receivePwrAllocationRequest`, it is collected by `drive3` agent on line 90 and the scenario ends gracefully.

2.3.4 Agent references

Agent references are used in definitions of action conditions, and represent single agent or groups of agents to whom messages are to be sent or from whom messages are to be received. All the agent references used in scenarios must be declared in the agent declaration section of SDLMAS file. A reference is characterized by its type (agent role), cardinality (single or group) and binding method. Five agent reference types are used within SDLMAS scenario definition: *verbatim*, *variable*, *anonymous*, *group* and *group variable* references.

Verbatim reference is fixed at scenario definition and denotes exactly one (named) agent. Line 76 of the example contains a message transmission condition that uses a verbatim reference; the message is sent specifically to an agent named "battery". Verbatim reference is usually used when referring to a one-of-a-role agent in the system (as "battery" agent is the only agent of type `PowerStorage` in the system).

Variable references, prefixed with $, are initially not bound to a particular agent and must be set 'internally' by the agent logic during a new conversation context (i.e. scenario instance) initialization. In `issuePwrAllocationRequest` action (line 11), an agent of type `Planner` initiates a `ConsumerPowerRequest` scenario by sending a `REQUEST` message to its coordinator agent. The coordinator agent must be determined by each planer agent separately, as not all `Planner` agents belong to the same coordinator.

Anonymous references, prefixed with @, are also initially not bound. In contrast to variable references, agent binding occurs externally and is performed by the framework itself during creation of a new conversation context. Line 15 contains an example of an anonymous reference usage, where binding occurs the moment a `REQUEST` message is received from one of the `Planner` agents and a new conversation context (scenario instance in the coordinator agent) is created.

Group references, enclosed within `< >`, denote all agents of a particular type (role) present in the system at the moment a conversation context is created. Group reference membership can change (decrease) as conversation with a particular group member can be terminated at any time during a scenario execution. At line 16 of the scenario example, a coordinator agent dispatches a `CFP` message to all of the agents playing the `SchedulerProxy` role, and at line 37 collects all the responses from those agents. Agents that have sent a `#REFUSE` message are removed from the group and the conversation with them is terminated automatically.

Group variable references, enclosed within `<$ >`, differ from group references only in the method of group population; while group references are populated externally (by the SDLMAS runtime), group variable references are populated by internal agent logic, the same as variable references. Group variable reference usage example can be found at line 22, where a scheduler proxy agent sends a `CFP` message to all the scheduler agents it coordinates and line 32 where responses are collected.

Variable and anonymous reference bindings exist as long as the enclosing conversation context exists. Multiple conversation contexts active on the same agent have distinct bindings of variable, anonymous, group and group variable references.

2.3.5 Performative types

One of the major challenges in managing conversations within a multi-agent system, apart from handling multiple and parallel conversation states, is to correctly detect when the conversation between two agents needs to be terminated. In some cases it can be implicitly realized from a scenario description, such as lack of σ in action definition, but in most cases it must be explicitly stated. Terminating performatives explicitly indicate an end of a conversation context and are prefixed with `#`. Terminating performatives both at sending and receiving conversation actions need to be appropriately marked. Non-terminating performatives are called conversational performatives and they preserve the active conversation context.

The simple example of a terminating performative can be found at line 17 (message reception at line 90), where coordinator informs planner agent that its request is accepted and no further conversation is necessary. More complex example involves a group of scheduler agents (line 32) individually returning either conversational `PROPOSE` (retain group membership) or terminating `#REFUSE` (evicted from the group) performative.

2.3.6 Modeling interactions with SDLMAS

In order to describe interactions within a multi-agent system using SDLMAS, we propose the following steps to be performed:

1. *Agent role identification.* Following the analysis of system interactions (usually available in the form of sequence diagrams or free text), a union of agent types (i.e. roles) at the system level has to be identified and respective agents and their types defined in SDLMAS.

2. *Generalization of interactions.* During the scenario definition process a number of generalizations can be introduced, mainly by substituting concrete agent names with anonymous, variable and group names. Generalized scenarios can be made more flexible and applicable in a wider number of cases (different pairs/groups of interacting agents), as well as scalable with respect to number of participating agents.

3. *Definition of conversation actions.* For each group of received or transmitted messages a definition of a conversation action is needed. Expressiveness of such actions is significantly higher than those of sequence diagrams, requiring definition of message transmission and reception constraints. Each action also must be provided a name of internal agent logic procedure invoked upon successful reception of all incoming messages.

4. *Terminating performatives.* All terminating performatives must be identified and appropriately marked.

2.4 Relation between SDLMAS and ACL

The SDLMAS, as a declarative language for scenario description, relies on the FIPA ACL [ACL] language in such way that the message aspect of the SDLMAS language is mapped into the ACL messages during code generation. The SDLMAS borrows some concepts from the ACL in order to make this mapping as seamless as possible. More specifically, it explicitly uses concept of performatives in scenario description, which will consequently be used to create ACL messages with appropriate message performatives. Few other aspects are also used within the SDLMAS platform. Conversation id is used for identification of particular conversation, especially in case of several simultaneous conversations. Sender and receiver parameters are used also to define corresponding agents according to scenario description while ontology parameter is used as a container for a communication of agent internal states ontology.

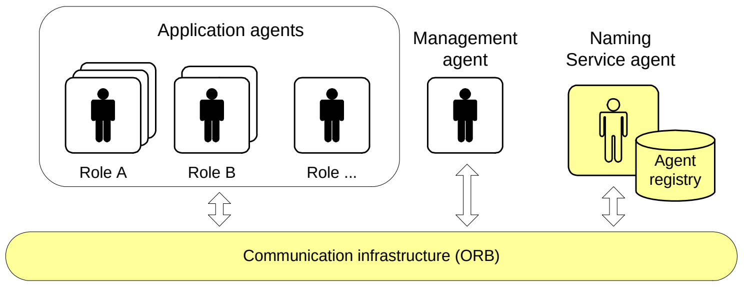


Figure 3: SDLMAS platform components

3 SDLMAS Platform

SDLMAS platform provides tools and a framework for implementation and run-time support of multi-agent systems whose interactions are modeled using SDL-MAS scenario description language.

3.1 Platform Components

A multi-agent system based on the SDLMAS platform consists of the three component groups: a generic SDLMAS component (Management agent), application-specific SDLMAS components (Application agents) and underlying agent platform components (ORB and Naming Service agent).

The SDLMAS platform components rely on resources and functionality provided by target agent platform, such as thread management and multithreaded execution of agents, FIPA compliant messaging etc. Naming service must also be provided by the target platform, and is used as a central registry of application agents currently active in the system. Upon their successful initialization, all agents are required to register with the Naming Service agent, and to deregister prior to their deactivation. Accurate information stored in the registry is crucial for correct behavior of late agent reference binding mechanism:

– Before an anonymous reference is bound to a real agent, an agent's type (role) is verified using the information from the registry.

– When a group reference is populated, all agents of the required type (role) are collected from the registry.

Management agent provides a support for bootstrapping and initialization of a multi-agent system based on provided global and agent-specific configurations. This agent is a mandatory system element.

Functionality of a multi-agent system is based on individual functionalities of system-constituting entities and on effects of interaction among those entities.

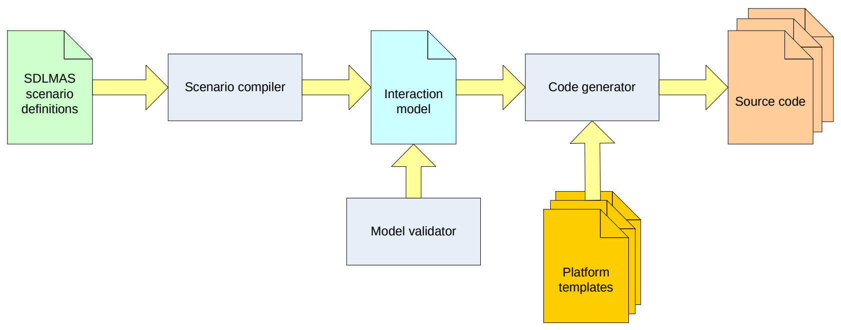


Figure 4: Code generation from scenario definition

Each SDLMAS application agent is assigned a specific role in the system and strictly conforms to prescribed interaction patterns for that role. Consequently, portions of the implementation code related to an interaction are generated from scenario descriptions, and must not be modified by agent developers. Other portions of agent code, related to internal agent logic, are free for developers to design and implement.

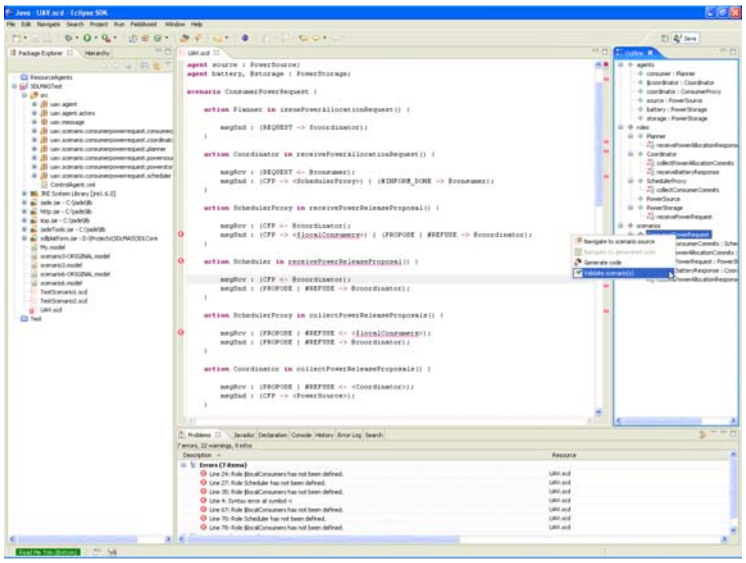
The following procedure is followed on system start-up: (1) Naming Service and Management agents are started, (2) a number of application agents are started by the Management agent (optional), (3) application agents register at the Agent registry (Naming Service agent), (4) a number of scenarios are initiated by the Management agent based on system configuration files, with specified application agents as their initiators.

3.2 Code Generation from Scenario definitions

The process of converting a SDLMAS scenario definition to a source code for a target agent platform is depicted on [Fig. 4].

A text file containing agent type declarations and scenario definitions is converted to an internal model, suitable for both scenario validation and code generation. Source code for the target agent platform is generated by employing the platform-specific set of code templates. This approach allows both easy re-targeting of generated agent code and requires only a modest effort in developing support for a new agent platform or programming language. Generated source entities are divided into two main categories: model-level entities, shared among many system components, and scenario-level entities, containing role- and scenario-specific implementation of agent communication behavior.

Two code generation tools have been developed for converting SDLMAS scenario definitions to platform code; a command line based tool and an Eclipse



```
agent source : PowerSource;
agent battery, $storage : PowerStorage;

scenario ConsumerPowerRequest {

    action Planner in issuePowerAllocationRequest() {

        msgSnd : (REQUEST -> $coordinator);
    }

    action Coordinator in receivePowerAllocationRequest() {

        msgRcv : (REQUEST <- $consumer);
        msgSnd : (CFP -> <SchedulerProxy>) | (#INFORM_DONE -> $consumer);
    }

    action SchedulerProxy in receivePowerReleaseProposal() {

        msgRcv : (CFP <- $coordinator);
        msgSnd : (CFP -> <$localConsumers>) | (PROPOSE | #REFUSE -> $coordinator);
    }

    action Scheduler in receivePowerReleaseProposal() {

        msgRcv : (CFP <- $coordinator);
        msgSnd : (PROPOSE | #REFUSE -> $coordinator);
    }

    action SchedulerProxy in collectPowerReleaseProposals() {

        msgRcv : (PROPOSE | #REFUSE <- <$localConsumers>);
        msgSnd : (PROPOSE | #REFUSE -> $coordinator);
    }

    action Coordinator in collectPowerReleaseProposals() {

        msgRcv : (PROPOSE | #REFUSE <- <Coordinator>);
        msgSnd : (CFP -> <PowerSource>);
    }
```

```
7 errors, 22 warnings, 0 infos
Description                                              Resource
Errors (7 items)
  Line 24: Role $localConsumers has not been defined.    UAV.acd
  Line 27: Role Scheduler has not been defined.          UAV.acd
  Line 35: Role $localConsumers has not been defined.    UAV.acd
  Line 4: Syntax error at symbol <                       UAV.acd
  Line 67: Role $localConsumers has not been defined.    UAV.acd
  Line 70: Role Scheduler has not been defined.          UAV.acd
  Line 78: Role $localConsumers has not been defined.    UAV.acd
```

Figure 5: Eclipse SDLMAS plug-in

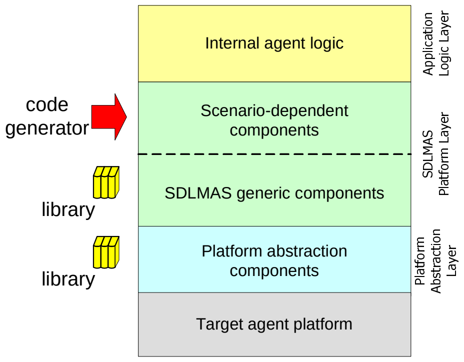


Figure 6: Layered agent internal structure

plug-in [Fig. 5]. The plug-in allows for easier syntax checking, model transformation and validation, as well as navigation between scenario definition and generated implementation code. Internal interaction model is stored in an EMF-based model [EMF] and JET templates are used for generating the Java code. At the moment only the JADE platform [JADE] is supported.

Three main layers can be identified within a SDLMAS agent implementation ([see Fig. 6]): platform abstraction, SDLMAS platform and application layers.

Platform Abstraction Layer is the layer closest to the target agent platform, abstracting the specificities of platform programming interfaces and semantics and providing the SDLMAS Platform layer with a unified interface towards the target platform resources. This layer takes the form of a library and is highly agent-platform and language dependant, thus must be developed for each new platform the SDLMAS is ported to.

SDLMAS Platform Layer consists of two sub layers. Generic components sub layer implements scenario- and role- independent functionality, and is provided in a form of a library, whereas Scenario-dependant sub layer originates from a process of code generation from SDLMAS scenario definitions. This sub layer contains necessary functionality for handling conversation contexts, tracking conversation progress, enforcing message reception and transmission conditions and invoking internal agent logic procedures.

Application Logic Layer encapsulates agent's application logic, and presents the only part of the agent's implementation where additional code must be added by developers. Application logic is implemented as a set of activities (invocations of method skeletons) asynchronously executed within a context of a scenario instance

3.3 Platform run-time behavior

Generating implementation code from SDLMAS scenario descriptions and providing a runtime support for multi-agent system execution basically requires a paradigm shift; from declarative interaction-centric approach of scenario definition to agent-centric approach of managing conversation states, active actions, message buffers etc. The following text presents only general concepts of platform runtime and depicts relations between language and runtime elements.

3.3.1 Scenario Execution

Scenario execution for an agent playing a scenario role takes a form of sequential execution of conversation actions belonging to that role. Order of those actions is determined by their relative position in the scenario definition.

Scenario can be triggered using two methods: external or internal. External method is used by the Management agent, and relies on sending a `SCENARIO_START` message to an agent who is to trigger the requested scenario. This approach does violate one of the key agent properties - autonomy, but is included due to its practical value, especially in bootstrapping the system or to achieve repetitive scenario execution. Internal scenario start method is used within agent logic implementation, i.e. an agent autonomously decides to trigger a certain scenario, possibly as a result of other scenario execution. To start a specific scenario, the agent must be of type (role) that contains a scenario triggering action. Scenario cannot be triggered using external method if its scenario triggering action uses agent references other than group or verbatim.

A conversation context denotes an independent scenario instance execution within an agent (context owner). It encapsulates all the elements forming the scenario instance state, i.e. the states of all the agent's conversations, current message reception and transmission conditions and the currently active conversation action. New conversation context can be created as a consequence of two different events. The first event is the activation of the scenario triggering action, regardless of the activation method used (external or internal). This conversation context is called a primary conversation context and is a parent context to all the conversation contexts created during the scenario instance execution. In this case the initiator and the owner of the context is the agent that triggered the scenario. A conversation context is also created when an agent receives a message that activates the first conversation action in a scenario for a role the agent plays. The initiator of this context is the agent that initiated the conversation (triggered the context creation).

In most cases, a conversation context is terminated (i.e. scenario instance execution within an agent ends) when the conversation between the context initiator and the context owner is terminated. Another cause of context termination can be the lack of message reception or transmission conditions in an active conversation action, preventing an agent to proceed with interaction.

3.3.2 Conversations

During the course of one scenario instance execution, an agent can be involved in many parallel conversations with other agents. At least one conversation is present, with an agent that initiated the conversation context (scenario instance). Each conversation, the agent is involved in, is characterized by a unique conversation identifier and its state maintained in a state variable.

Conversation state change can only occur within the context of a conversation action, as a consequence of a conversation message being received and/or transmitted. If received, transmitted or both messages are conversational (i.e. contain conversational performatives), the conversation state is changed after the conversation action finishes. However, if received or transmitted message is a terminating message (i.e. contains a terminating performative), conversation state is immediately changed to 'exit' and the conversation is terminated. Conversation can also be moved to 'exit' state if the executed action is the last scenario action for a particular role the agent is playing. [Fig. 7] depicts state transitions in a simple ContractNet example where an `Inititator1` request proposals from two participants. `Initiator1` creates a conversation context containing two independent conversations with participants. The conversations will progress in parallel until one of them is terminated by `#REJECT` message, sent by `Initiator1`. `Initiator1` will remove conversation with `Participant2` from

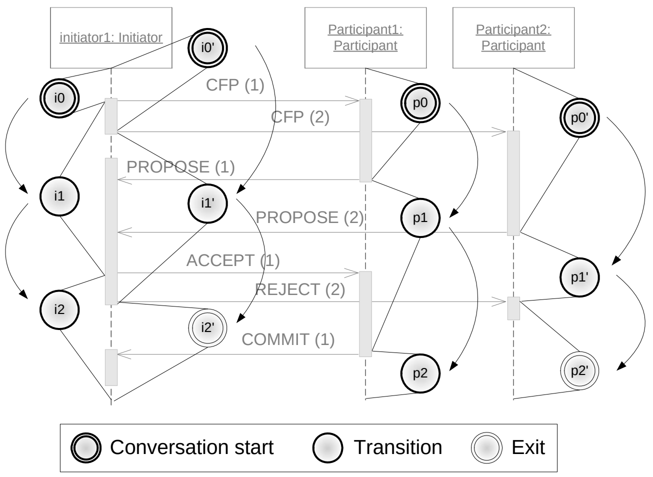


Figure 7: Conversation states

its own conversation context and continue only with `Participant1`. Similarly, conversation context on `Participant2` side will also be destroyed.

Pronouncing a conversation as terminated is not the only step that needs to be taken; scenario conversation actions may still enforce message reception and transmission conditions including atomic conditions involving agents that are no longer reachable. Without conducting the constraint elimination procedure on all existing reception and transmission conditions currently defined within the scenario instance, the system would end in a deadlock or its execution being seriously affected by a large number of timeouts. An example of a condition elimination procedure is presented on [Fig. 8], where a transmission condition is eliminated in `Participant2` agent (received a terminating message from `Initiator1`), and a reception condition is eliminated in `Initiator1` agent (terminated conversation with `Participant2` by sending him a `#REJECT` message).

3.3.3 Conversation Actions

Conversation actions are atomic units of scenario execution. Each agent performing a certain role is characterised by a sequence of scenario-specific conversation actions. As defined in the SDLMAS language specification, for an action to be triggered a set of messages conforming to the action's reception conditions must be received. As a consequence of successful action execution, the messages are sent to various scenario participants and the agent's current conversation action is changed.

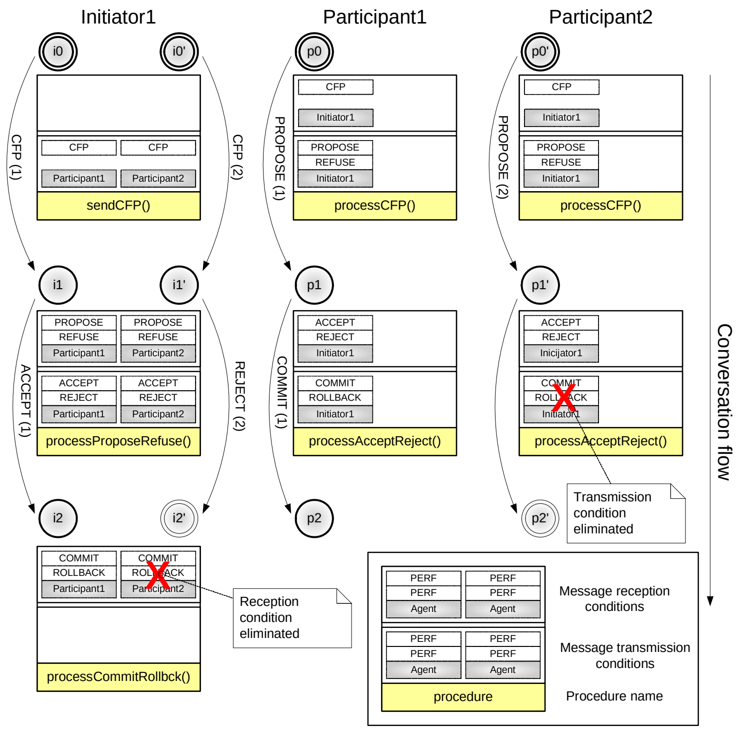


Figure 8: Constraint elimination

Message reception process is a three-step process involving scenario instance routing, message expectance verification and conversation action triggering. In the first step, an active scenario instance within an agent is selected according to the scenario identifier field value of the received message, and the message is forwarded to its message handling subsystem. In the second step, a message expectance is determined according to the basis of several factors:

– current scenario conversation action defining message reception conditions,

– current conversation state,

– message originating agent,

– message performative.

If there is at least one atomic reception condition within the current conversation action's message reception condition whose source and type (performative) match the received message's source and type, the message is considered to be expected. Expected messages are forwarded to conversation action's message buffer, while non-expected messages are discarded.

The third step, triggered by a buffer message insertion, involves checking the message reception constraint satisfaction. If the set of messages currently present in the conversation action's message buffer satisfies the action's compound reception constraint, conversation action execution is triggered. A set of messages satisfying the constraint is forwarded to the action procedure implementation along with an outgoing message template. The SDLMAS platform does not prescribe any constraints on implementation of conversation action procedures, just expects that all the outgoing messages, according to the forwarded template, are properly formed and they conform to message transmission conditions.

A timeout mechanism is invoked if the active action's reception conditions are not satisfied for a specified time period. When timeout occurs, all expected but not received messages are provided by the platform run-time with the `#TIMEOUT` terminating performative set, and the conversation action execution is triggered.

Message transmission conditions determine a set of allowable combinations of outgoing messages (characterized by their performatives) and their receivers. No explicit consistency checking mechanisms are employed; an implicit one is used by providing an outgoing message template structure during procedure invocation. Agent logic implementers are advised to use provided outgoing message templates. Outgoing message template contains an allowed set of outgoing messages with a limited number of selectable performatives and preset receiver agents, valid at the point of scenario execution.

3.3.4 Security

Providing security in multi-agent systems does not require addressing of only standard security issues present in multi-user distributed systems, but several MAS-specific issues as well [FIPA 01] [Poggi et al. 01] [Novák et al. 03]. Existing multi-agent platforms generally combine two approaches to usage of security mechanisms with regards to agent implementation. *Transparent* mechanisms are implemented at the infrastructural level and do not influence the implementation of agents forming the multi-agent application. In contrast, *opaque* mechanisms are used to implement security at the application level and require explicit usage of security API within agent implementation code.

The SDLMAS platform security and security of applications built on top of the SDLMAS platform rely solely on security mechanisms provided by the target agent platform or its security extensions. For example, SDLMAS for JADE as a target agent platform requires JADE Security add-on [JADE-S] to provide security mechanisms for the developed application. Basic security is achieved using only transparent security mechanisms provided by the add-on: principal-based resource usage authorization and secure communication among platform containers is configured using respective configuration files, requiring no changes in the original SDLMAS for JADE platform code or in implementation of application agents.

However, such an approach does not resolve security issues related to integrity and/or confidentiality of exchanged messages and authorized usage of SDLMAS platform security-critical services. SDLMAS platform contains three such security-critical services: (1) agent registration/deregistration with the Naming Service agent, (2) adding/revoking group change notification subscriptions and (3) external scenario triggering. To resolve mentioned issues, it is necessary to employ opaque target platform security mechanisms within a SDLMAS platform implementation for signing and/or encryption of exchanged messages.

We are in the process of deriving a secure SDLMAS for JADE-S platform implementation from existing SDLMAS for JADE platform by using JADE-S provided opaque security mechanisms. Required modifications are confined to the Platform Abstraction Layer, where all the messages shall be signed and encrypted automatically. In this way we will, transparently to the application agent implementation layer, ensure integrity and confidentiality of exchanged agent messages, and enforce authorized access to security-critical platform services (for example, only the Management agent will be allowed to externally trigger scenarios).

In our future work we plan to address fine-grained communication security among agents by allowing declaration of security-related information within scenario descriptions in concert with adding target platform-independent security sub layer within the SDLMAS Platform layer.

4 Related Work

In the last few decades, numerous methodologies for development of agent-based systems appeared as a part of an initiative to formalize and define the process of design and implementation of multi-agent systems. Agent interactions are not usually treated separately as they are seen as an ingredient element of activities like design, development or deployment of such systems. In general, the majority of models [Gmez-Sanz et al. 03] [Wood et al. 00] [Wooldridge et al. 00] are based on AUML [AUML], Petri Nets [Cost et al. 99] or state-chart diagrams. Although AUML achieved significant level of popularity, generally for its visual representation, it lacks the expressiveness required due to the multi-lateral nature of agent interactions [Paurobally et al. 03a]. Additionally, the diagrams can become hard to read if they need to describe slightly more complicated conversations with several agents involved. Also, they do not include mapping definitions between interaction protocols and agent's internal actions which makes this language impractical for rapid agent development by automatic code generation.

Petri Nets are also not very appropriate for interaction description due to their lack of clarity and scalability [Richters et al. 98] [Paurobally et al. 03b], especially for protocols like Contract-Net [Purvis et al. 02]. State-charts have clearer representation of interaction protocols but they are still missing clear definition of relationship between protocol execution and an agent's business logic.

The model described in [Bratman et al. 88] is based on a belief-desire-intention (BDI) scheme where an agent tries to fulfill its own goals by executing tasks based on own knowledge base about environment. Although the model influenced definitions of widely accepted FIPA and ACL [FIPA] standards, it suffers from the "semantic verification" problem [Wooldridge 00] where it is not possible to guarantee the equal understanding of ontology terms by all agents involved.

The MAP language [Walton 03] for dialogue protocol definition is based on the principles found in Electronic Institutions [Estava et al. 01] while its semantics is inspired by logic which is basis for communication and parallel systems [Milner 89]. The key concept of the language is the decomposition of a dialogue into scenes. A scene can be seen as an interaction context within which agents are communicating with each other. A scene also contributes to a better organization of complex multi-agent systems by enabling start of interactions only when all agents are present within a given scene, or, by preventing agents to leave a scene before a dialogue completes. Another key concept is an agent's role, a certain set of behaviors that an agent will adopt during interactions. An agent adopts an interaction protocol based on the role it plays. The language also allows agents to interact asynchronously and simultaneously which might cause unforeseen system activities, even if a well-structured protocol is introduced in order to guarantee system's emergent behavior. A potential solution to this problem lies in the exhaustive search of the entire dialogue space in order to check potential conflicting protocol definitions, such as loops or deadlocks during interactions [Clarke et al. 99]. It is important to note that this language is not based on message flows.

Another attempt to represent agent interactions and roles in a standardized way resulted in the appearance of AgentUML (or AUML) [Huget 04]. The UML standard is taken as the base in order to mitigate a paradigm shift from object-oriented to agent-based concepts, and to enable standardized notation for analysis, design and implementation of agent systems. UML class diagram is amended in order to include concepts of roles and behaviors while sequence diagram is extended by specificities of agent interactions. Visual representation of agent dialogues is one of the advantages of the AgentUML. In order to achieve practical usability, it is necessary to define the ways of its transformation into textual notation of protocols, and various language transitions are proposed [Warmer et al. 99] [Winikoff 05]. In spite of these efforts, AUML lacks enough representation capabilities of agent's states, which causes an inability to define conditions under which messages can be received or sent by an agent. Although AUML lacks some features mentioned above, it has influenced other language designs [Warmer et al. 99] [Winikoff 05] [Dinkloh et al. 05] [Huget 02] and modeling frameworks [Quenum et al. 06]. The OCL language [Warmer et al. 99] is the way to represent UML in a textual format and it can be used to represent AUML too but it was shown to have limited usability in [Richters et al. 98].

The IOM/T language [Doi et al. 05] introduces a formal way of mapping interaction protocol from AUML and in general emerged from a tendency to strictly define protocols in a textual notation. Unlike the MAP language, the language is based on message flow which means the definitions of protocol-related agent activities are not separated but defined in the single definition. Unfortunately it lacks some important characteristics related to scenario descriptions.

Firstly, there is no explicit definition of message performative used in conversations. A protocol is defined as a sequence of messages whose character is determined by corresponding performatives. If the language lacks formal explicit definition of message performatives, message validity needs to be performed within the agent's business logic. This increases the complexity of the logic implementation with additional performance penalties since irrelevant or protocol-incompatible messages will not be immediately ignored after their arrival but during their processing. Furthermore, it is not possible to determine the differences between conversational and terminating performatives which prevents an agent to explicitly recognize a conversation completion. Since there is no clear distinction between performative types, it is also not possible to define conditions under which messages can be forwarded to the business logic or just ignored.

Secondly, the cardinality of agent instances is bound to the protocol implementation inside of agent's internal logic. The protocol description is linked with the given scenario and the number of agent instances in a dialogue. If it is required to change a cardinality of agent instances, the protocol description needs to be modified too in order to reflect this update since the business logic defines the interaction description including agent instance cardinality. Consequently, it is not possible to change number of instances of a particular agent without re-implementing the internal implementation.

Thirdly, an implementation of agent's internal logic is language-dependent on the JADE agent platform [JADE], which disables possibility of usage IOM/T language in another agent platform.

Q Language [Ishida 02] is another language for interaction protocol definition in a textual form. Its main purpose is to define interactions with a user or other agents on behalf of a user. Consequently, it is not based on message flows in the way that SDLMAS is. Typically, scenario description will be considered only from the point of view of one agent who is supposed to interact with other parties assuming their prior knowledge about the scenario they are supposed to follow.

5 Conclusions and Future Work

In this paper we presented the SDLMAS framework for rapid design, development and runtime support of multi-agent systems. The framework consists of a declarative language for description of interactions within a multi-agent system, tools for description and model manipulation, and a runtime framework.

The presented language is a declarative one, describing external system behavior in a form of a set of interaction scenarios, where each scenario describes an interaction episode (a trace of communication) among a group of agents. The interaction-centric perspective the SDLMAS language takes proves to be far simpler and more intuitive than agent-centric approaches, yet providing enough expressiveness, flexibility and scalability in describing complex and interwoven interactions.

Scenario descriptions are transformed into a target agent platform's program code, and in conjunction with target platform and platform-specific SDLMAS runtime support, form a basis of a functional multi-agent system. This transformation of interaction-centric scenario descriptions into agent-centric implementation code required both a paradigm shift and a careful mapping of language constructs to runtime constructs, in order to avoid inconsistencies between declared and runtime system behavior.

By using SDLMAS, an effort invested in design and implementation of interaction related portions of multi-agent systems is significantly reduced, allowing more resources to be dedicated to higher functional aspects of developed systems.

One of the current limitations of the SDLMAS framework is a lack of explicit support for loops within scenario descriptions. A language construct for specifying timing aspects of message reception constraints would also contribute to usability of the framework. These limitations, in conjunction with an effort to support agent platforms other than JADE, will be addressed in our future work as a part of continuous improvement of the SDLMAS platform.

Acknowledgements

This work has been supported by the Croatian Ministry of Science, Education and Sports under the research project "Software Engineering in Ubiquitous Computing".

References

[ACL] FIPA Agent Communication Language. See `http://www.fipa.org/repository/aclspecs.html`.

[AUML] AgentUML. See `http://www.auml.org`.

[Bratman et al. 88] Bratman, M. E., Israel, D. J., Pollack, M. E.: "Plans and Resource-Bounded Practical Reasoning"; Computational Intelligence, 4 (1988), 349-355.

[Clarke et al. 99] Clarke, E. M., Grumberg, O., Peled, D. A.: "Model Checking"; MIT Press, Cambridge (1999).

[Cost et al. 99] Cost, R., Chen, T., Finin, T., Labrou, Y., Peng, Y.: "Modeling Agent Conversations With Colored Petri Nets"; Proc. Workshop on Specifying and Implementing Conversation Policies, Seattle, USA (1999), 59-66.

[Dinkloh et al. 05] Dinkloh, M. Nimis, J.: "A Tool for Integrated Design and Implementation of Conversations in Multiagent Systems"; Proc. AAMAS03 PROMAS Workshop on Programming Multi-Agent Systems Selected Revised and Invited papers, Melbourne, Australia (2003), 187-200.

[Doi et al. 05] Doi, T., Tahara, Y., Honiden, S., "IOM/T: An Interaction Description Language for Multi-Agent Systems"; Proc. 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05), Utrecht, Netherlands (2005), 778-785.

[EMF] The Eclipse Modeling Framework; See `http://www.eclipse.org/modeling/emf/`

[Endriss et al. 03] Endriss, U., Maudet, N., Sadri, F., Toni, F.: "Protocol Conformance for Logic-based Agents", Proc. 18th International Joint Conference on Artificial Intelligence (IJCAI-2003), Acapulco, Mexico (2003), 679-684.

[Estava et al. 01] Estava, M., Rodriguez, J. A., Sierra, C., Garcia, P., Arcos, J. L.: "On the Formal Specifications of Electronic Institutions"; In Agent Mediated Electronic Commerce, The European AgentLink Perspective, Lecture Notes In Computer Science vol. 1991., Springer-Verlag, London (2001), 126-147.

[FIPA] FIPA: Foundation for Intelligent Physical Agents. See `http://www.fipa.org`

[FIPA 01] FIPA Agent Security Management Specification (obsolete). See `http://www.fipa.org/specs/fipa00020/index.html`

[Gmez-Sanz et al. 03] Gmez-Sanz, J. J., Fuentes, R.: "Agent Oriented Software Engineering with INGENIAS"; Proc. 3rd International Central and Eastern European Conference on Multi-Agent Systems CEEMAS 2003, Prague, Czech Republic (2003), 394-403.

[Greaves et al. 00] Greaves, M., Holmback, H., Bradshaw, J., "What Is a Conversation Policy?"; In Issues in Agent Communication, F. Dignum and M. Greaves, Eds. Lecture Notes In Computer Science, vol. 1916. Springer-Verlag, London, UK (2000), 118-131.

[Huget 02] Huget, M. P.: "A Language for Exchanging Agent UML Protocol Diagrams"; Technical Report ULCS-02-009, The University of Liverpool, Computer Science Department, UK (2002).

[Huget 04] Huget, M. P.: "Agent UML Notation for Multiagent System Design"; IEEE Internet Computing, 8, 4 (2004), 63-71.

[Ishida 02] Ishida, T., Q: "A Scenario Description Language for Interactive Agents"; Computer, 35, 11 (2002), 42-47.

[JADE] JADE: Java Agent Development Framework. See `http://jade.cselt.it`

[JADE-S] JADE Security add-on. See `http://jade.tilab.com/doc/tutorials/JADE_Security.pdf` (2005)

[Jennings et al. 99] Jennings, N. R., Wooldridge, M.: "Agent-Oriented Software Engineering", Proc. 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Multi-Agent System Engineering (MAAMAW-99), Valencia, Spain (1999), 1-7.

[Milner 89] Milner, R., "Communication and Concurrency"; Prentice-Hall International (1989).

[Novák et al. 03] Novák, P., Rollo, M., Hodík, J., Vlček, T.: " Communication Security in Multi-agent Systems"; Proc. 3rd International / Central and Eastern European Conference on Multi-Agent Systems CEEMAS 2003, Prague, Czech Republic (2003), 454-463.

[Papazoglou 01] Papazoglou, M. P.: "Agent-oriented technology in support of e-business"; Communications of the ACM, 44, 4 (2001), 71-77.

[Paurobally et al. 03a] Paurobally, S., Cunningham, J.: "Achieving Common Interaction Protocols in Open Agent Environments"; Proc. 2nd international workshop on Challenges in Open Agent Environments (AAMAS 03), Melbourne, Australia (2003).

[Paurobally et al. 03b] Paurobally, S., Cunningham, J., Jennings, N. R.: "Developing Agent Interaction Protocols Using Graphical and Logical Methodologies"; Proc. AAMAS03 PROMAS Workshop on Programming Multi-Agent Systems Selected Revised and Invited papers, Melbourne, Australia (2003), 149-168.

[Podobnik et al. 08] Podobnik, V., Petric, A., Jezic, G.: "Agent-Based Solution for Dynamic Supply Chain Management"; Journal of Universal Computer Science, 14, 7 (2008), 1080-1104.

[Poggi et al. 01] Poggi, A., Rimassa, G., Tomaiuolo, M.: " Multi-user and security support for multi-agent systems"; Proc. WOA 2001, Modena, Italy (2001), 13-18.

[Purvis et al. 02] Purvis, M. K., Cranefield, S., Nowostawski, M., Ward, R., Carter, D., Oliviera, M. A., "Agent Cities Interaction Using the Opal Platform"; Proc. Workshop on Challenges in Open Agent Systems, The 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS02), Bologna, Italy (2002).

[Quenum et al. 06] Quenum, J. G., Aknine, S., Briot, J-P, Honiden, S.: "A Modelling Framework for Generic Agent Interaction Protocols"; Proc. 4th International Workshop on Declarative Agent Languages and Technologies, Hakodate, Japan (2006), 207-224.

[Richters et al. 98] Richters, M., Gogolla, M.: "On Formalizing the UML Object Constraint Language"; Proc. 17th International Conference on Conceptual Modeling, Singapore (1998), 449-464.

[Sliwko et al. 07] Sliwko, L., Nguyen, N. T.: "Using multi-agent systems and consensus methods for information retrieval in internet"; International Journal of Intelligent Information and Database Systems, 1, 2 (2007), 181-198.

[Stranjak et al. 08] Stranjak, A., Dutta, P. S., Ebden, M., Rogers, A., Vytelingum, P.: "A Multi-Agent Simulation System for Prediction and Scheduling of Aero Engine Overhaul"; Proc. 7th International Conference on Autonomous Agents and Multiagent Systems: industrial track (AAMAS2008), Estoril, Portugal (2008), 81-88.

[Walton 03] Walton, C. D.: "Multi-Agent Dialogue Protocols"; Proc. 8th International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida (2003).

[Warmer et al. 99] Warmer, J., Kleppe, A.: "OCL: The Constraint Language of the UML"; Journal of Object-Oriented Programming (1999), 10-13.

[Winikoff 05] Winikoff, M.: "Towards Making Agent UML Practical: A Textual Notation and a Tool"; Proc. 5th International Conference on Quality Software (QSIC'05), Melbourne, Australia (2005), 401 - 412.

[Wooldridge 00] Wooldridge, M.: "Semantic Issues in the Verification of Agent Communication Languages"; Autonomous Agents and Multi-Agent Systems, 3, 1 (2000), 9-31.

[Wood et al. 00] Wood, M. F., DeLoach, S. A.: "An Overview of the Multiagent Systems Engineering Methodology"; Proc. 1st International Workshop on Agent-Oriented Software Engineering, Limerick, Ireland (2000), 207-221.

[Wooldridge et al. 00] Wooldridge, M., Jennings, N. R., Kinny, D.: "The Gaia Methodology for Agent-Oriented Analysis and Design"; Autonomous Agents and Multi-Agent Systems Archive, 3, 3 (2000), 285-312.

[Zhang et al. 08] Zhang, D., Simoff, S., Aciar, S., Debenham, J.: "A multi agent recommender system that utilises consumer reviews in its recommendations"; International Journal of Intelligent Information and Database Systems, 2, 1 (2008), 69-81.